SENG1120 – Data Structures
Trimester 2, 2024

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Assignment	*2*	**Due Date**	*Sunday 14 July @ 23:59*
Purpose	*To measure the student's ability to implement and/or use one or more specialized containers (e.g., stacks and queues) to solve an underlying problem in C++.*		

Overview

This assignment will test your ability to implement a (linked) queue as well as usage of its functionality to simulate a queue of patients at a doctor's office. In summary, you will build a simulation of a queuing system where there are a set number of available doctors and patients that arrive are either allocated to a doctor or placed into a queue to be seen when a doctor becomes free.

We will model a system that primarily consists of a doctor's office (i.e., a collection of doctors) and a queue holding the patients waiting to be seen. Initially, all doctors are free and, hence, the first patient to arrive will immediately be seen by the first doctor. When the next patient arrives, they will immediately be seen if there is an available doctor. Otherwise, the patient is placed in a queue to wait. To appropriately model the queueing system of a doctor's office, we will need to know the number of doctors, the length of an appointment, and the average time between the arrivals of patients. For simplicity, the length of an appointment will be fixed for all patients.

This assignment is worth 100 marks and accounts for 10% of your final grade. Late submissions are subject to the rules specified in the Course Outline.

The Supplied Files

This section gives an overview of the files that you are provided as part of this assignment. **None of the provided files should be modified.** In contrast to Assignment 1, you are not provided with (skeleton) implementation files – you will be expected to create these files from scratch.

You are recommended to take some time to understand the overall structure of the program before starting the implementation phase. **Note:** In some cases, there are methods implemented in the supplied header files – this is done so that you don't have to worry about implementing them. You will be expected to use these methods so be sure to check them out!

- `main.cpp` – contains the main function, including the logic to parse the arguments, start the simulation, then display some statistics at completion.

- `empty_collection_exception.h` – contains the definition of an exception that can be thrown when an operation is attempted on an empty queue. **Note:** there is no accompanying `.cpp` file – the `.h` file contains the complete definition and implementation of the exception.

- `lqueue.h` – contains the header for the `LQueue` class, which includes the instance variables and behaviours that your queue should contain.

- `patient.h` – contains the header for the `patient` class, which includes the instance variables and behaviours that a patient should contain.

- `doctor.h` – contains the header for the `doctor` class, which includes the instance variables and behaviours that a doctor should contain.

- `doctors_office.h` – contains the header for the `doctors_office` class, which includes the instance variables and behaviours that the doctor's office should contain.

- `simulation.h` – contains the header for the `simulation` class, which includes the instance variables and behaviours that your simulation should contain.

- `makefile` – the `makefile` for the project, which outlines the build recipe. The `-g` flag is set for you to make debugging and/or memory leak detection more convenient, should you wish to use these tools.

Running the Program

Of course, you will need to ensure the program is compiled prior to running it. You can use the supplied `makefile` for this – it will produce an executable called `Simulation`. The program should be executed using the command:

```
./Simulation <seed> <sim_time> <num_doctors> <appointment_time>
<time_between_arrivals>
```

This means, you need to supply various integer arguments to the program for it to run. For example, you can run it using:

```
./Simulation 0 10 1 4 3
```

To start a simulation with a random seed of 0, 10 units of time, 1 doctor, an appointment time of 4 units, and an average time between patients of 3 units. You are not responsible for parsing of the arguments, nor are you expected to make the program work with incorrect arguments – your program will only be tested with valid integer inputs.

The output of this simulation should look like the following:

```
Patient 0 arrived at time 0
Doctor 0 seeing Patient 0 at time 0
Patient 1 arrived at time 2
Patient 2 arrived at time 3
Doctor 0 (Patient 0) done at time 4
Patient 3 arrived at time 4
Doctor 0 seeing Patient 1 at time 4
Patient 4 arrived at time 7
Doctor 0 (Patient 1) done at time 8
Doctor 0 seeing Patient 2 at time 8
Total waiting time: 14
Number of patients that completed an appointment: 2
Number of patients still being seen by a doctor: 1
The number of patients left in queue: 2
Average waiting time: 2.80
```

If we increase the number of doctors to 2 by running the command:

```
./Simulation 0 10 2 4 3
```

we should see the average wait time is decreased significantly:

```
Patient 0 arrived at time 0
Doctor 0 seeing Patient 0 at time 0
Patient 1 arrived at time 2
Doctor 1 seeing Patient 1 at time 2
Patient 2 arrived at time 3
Doctor 0 (Patient 0) done at time 4
Patient 3 arrived at time 4
Doctor 0 seeing Patient 2 at time 4
Doctor 1 (Patient 1) done at time 6
Doctor 1 seeing Patient 3 at time 6
Patient 4 arrived at time 7
Doctor 0 (Patient 2) done at time 8
Doctor 0 seeing Patient 4 at time 8
Total waiting time: 4
Number of patients that completed an appointment: 3
Number of patients still being seen by a doctor: 2
The number of patients left in queue: 0
Average waiting time: 0.80
```

Note: the random seed will change the arrival time of patients. Running the simulation with the same seed (and all other parameters the same) will produce the same output, while running the simulation with a different seed (and all other parameters the same) will likely produce different output, though not always. For example, running the simulation above with 1 doctor, and a seed of 1120, leads to an average wait time of 2.33. This is because the seed controls the sequence of numbers that are generated as "random" – we use this to reproduce the output and make testing easier.

Note: the program will not compile as supplied, as you will not have the necessary implementation files. You are encouraged to write skeleton files, similar to those in Assignment 1, to allow compilation of an incomplete program – this will allow you to work incrementally.

The Tasks

The first task is to implement templated queue class using a linked list as the underlying data collection. As the purpose of this assignment is to focus on the behaviour of the queue data structure, you are expected to use `std::list` (std::list - cppreference.com) to provide the linked list behaviour. However**, you are not permitted to use `std::queue`**. This avoids being double penalised if your linked list did not work correctly in Assignment 1 and provides valuable experience working with an existing collection from the standard template library.

Next, you will implement the various data classes, namely `patient`, `doctor`, and `doctors_office` that support the simulation. Then, you will implement the `simulation` class, which executes the simulation. Further details about each class are given below.

LQueue

The `LQueue` class is a templated queue and should be implemented as discussed in lecture, noting that you are required to use (a pointer to) `std::list` as the underlying list. For a full list of methods required and some important details, you should examine the `lqueue.h` file and its associated documentation.

patient

Each `patient` has a number, arrival time, waiting time, appointment length, and departure time. If we know the arrival time, waiting time, and appointment length, we can determine their departure time by adding these together!

The basic operations that must be performed are as follows: set the patient number, arrival time, and waiting time; increment the waiting time by one time unit; return the waiting time; return the arrival time; return the appointment time; and return the patient number. For a full list of methods required and some important details, you should examine the `patient.h` file and its associated documentation.

doctor

At any given time, a `doctor` is either busy serving a patient or is free – we will use a Boolean variable to indicate their status. The doctor also records the information of the patient currently being seen, including an indication of the time remaining for the current appointment.

Some of the basic operations that must be performed by a doctor are as follows: check whether the doctor is free; set the doctor as free; set the doctor as busy; set the appointment time (that is, how long it takes to see the patient); return the remaining time in the current appointment (to determine whether the doctor should be set to free); if the doctor is busy after each time unit, decrement the remaining time in the current appointment by one time unit; etc. For a full

list of methods required and some important details, you should examine the `doctor.h` file and its associated documentation.

doctors_office

The `doctors_office` is a wrapper that represents a vector of `doctor` objects. For the `patient` at the front of the queue, we need to find a doctor in the list that is free. If all the doctors are busy, then the patient must wait until one of the doctors becomes free.

Some of the operations that must be performed are as follows: return the doctor number of a free doctor; set the doctor to busy when they are seeing a patient; return the number of busy doctors; update the doctor list – for each busy doctor, reduce the remaining time in their current appointment by one time unit. If the remaining appointment time of a doctor becomes zero, set the doctor as free – this should also display an appropriate message (using `display_appointment_done(…)`). For a full list of methods required and some important details, you should examine the `doctors_office.h` file and its associated documentation.

simulation

The `simulation` is the driver of the simulation and handles the main queueing system logic. To run the simulation, we need to know the length of an appointment and the average time between patient arrivals. For simplicity, we will fix the length of an appointment, which we will supply as a program argument when running our simulation. For the patient arrivals, we will use a random number and Poisson distribution to determine whether a patient arrives at each time step – this adds some randomness to our simulation! Don't worry, the code to do this is already provided for you and you should use the function `has_patient_arrived(…)` to determine if a patient arrives at the current time step. **Note:** the argument passed to `has_patient_arrived(…)` function should be `time_between_arrival`, not the current time.

The bulk of the effort here will be in implementing the `run_simulation(…)` function. The general algorithm for this function is as follows:

```
for (int time = 0; time < sim_time; time++)
{
```

- Update the `doctors_office` (using `update_doctors(…)`) to decrement the remaining appointment time of each busy doctor by one time unit.
- If the patient queue is non-empty, increment the waiting time of each patient by one time unit. **Note**: you'll have to get clever here since you don't have access to the individual entries in the queue!
- If a patient arrives (i.e., `has_patient_arrived(…)` returns `true`), display a message (using `display_patient_arrived(…)`), increment the number of patients by 1, add the new patient to the queue.
- If there is a free doctor and the patient queue is non-empty, remove a patient from the front of the queue, increment the total waiting time, and send the patient to the free doctor. Display an appropriate message (using `display_patient_seen(…)`).

```
}
```

Marking

As mentioned previously, this assignment is worth 100 marks and accounts for 10% of your final grade. Late submissions are subject to the rules specified in the Course Outline.

Your submission will be assessed on both correctness and quality. This means, in addition to providing a correct solution, you are expected to provide readable code with appropriate commenting, formatting, best practices, memory management, etc. **Code that fails to compile will result in a zero for the functionality correctness section.** Your code may be tested on functionality not explicitly shown in the supplied demo file (i.e., using a different `main.cpp` file). Hence, you are encouraged to test broader functionality of your program before submission.

Marking criteria will accompany this assignment specification and will provide an indicative guideline on how you will be evaluated. **Note**: the marking criteria is subject to change, as necessary.

Submission

Your submission should be made using the Assignment 2 link in the Assignments section of the course Canvas site. Assignment submissions will not be accepted via email. Incorrectly submitted assignments may be penalised.

Your submission should include only the completed versions of `lqueue.hpp`, `patient.cpp`, `doctor.cpp`, `doctors_office.cpp`, and `simulation.cpp`. You do not need to include any other files in your submission. Be sure that your code works with the supplied files. Do not change the files we have supplied as your s ubmission will be expected to work with these files – when marking your code, we will add the required files to your code and compile it using the supplied `makefile`.

Compress the required files into a single `.zip` file, using your student number as the archive name. Do not use `.rar`, `.7z`, `.gz`, or any other compressed format – these will be rejected by Canvas. For example, if your student number is c9876543, you would name your submission:

c9876543.zip

If you submit multiple versions, Canvas will likely append your submission name with a version number (e.g., **c9876543-1.zip**) – this is not a concern, and you will not lose marks.

Remember that your code will conform to C++ best practices, as discussed in lecture, and should compile and run correctly using the supplied `makefile` in the standard environment (i.e., the Debian virtual machine). **If you have developed your solution using any other environment, be sure to test it with the VM prior to submission.** There should be no segmentation faults or memory leaks during or after the execution of the program. Please see the accompanying marking criteria for an indicative (but not final) guideline to how you will be evaluated.

Helpful Tips

A few tips that may help you along your journey.

1. Read the header files carefully – they provide further information on the specification for various methods.

2. Work incrementally – don't try to implement everything at once. Consider getting a barebones version to compile, which will enable you to test methods as you implement them.

3. Remember that you can debug your program in VS Code. See Lab3a for a brief guide. Of course, the name of the executable in your `launch.json` file will be different than the example in the lab, but this gives you a great way to inspect your data structures during the program execution. You will need to think carefully about what you would expect the data to look like.

4. You will be expected to supply a program with no memory leaks. You are encouraged to use `valgrind` to assess whether you have any leaking memory in your program.

5. Start early! The longer you wait to start, the less time you will have to complete the assignment.

Good Luck!